08 | 内存管理：Spark如何使用内存？

2021-09-27 吴磊

《零基础入门Spark》 课程介绍 >

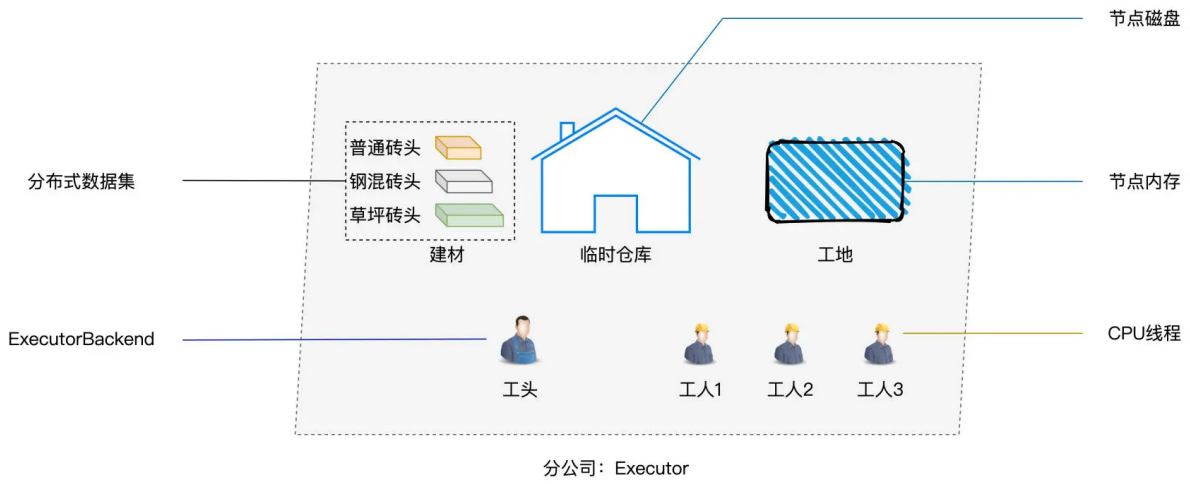
讲述：吴磊

时长 18:23 大小 16.84M

你好，我是吴磊。

在第 6 讲，我们拜访了斯巴克建筑集团的分公司，熟悉了分公司的办公环境与人员配置，同时用“工地搬砖的任务”作类比，介绍了 Spark Shuffle 的工作原理。

今天这一讲，我们再次来到分公司，去看看斯巴克公司都在承接哪些建筑项目，以及这些项目是如何施工的。通过熟悉项目的施工过程，我们一起来学习 Spark 的内存管理。



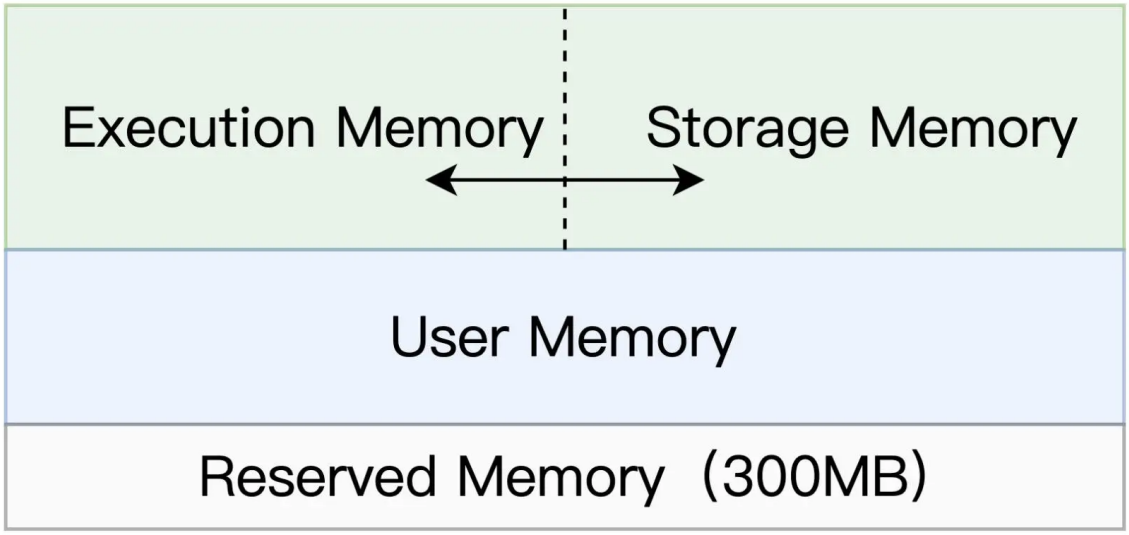
斯巴克建筑集团分公司

相比其他大数据计算引擎，关于 Spark 的特性与优势，想必你听到最多的字眼，就是“内存计算”。**合理而又充分地利用内存资源**，是 Spark 的核心竞争力之一。因此，作为开发者，我们弄清楚 Spark 是如何使用内存的，就变得非常重要。

好啦，闲言少叙，请你戴好安全帽，跟我一起再次去拜访斯巴克集团分公司吧。不过，在正式“拜访”之前，我们还有一项准备工作要做，那就是先了解清楚 Spark 的内存区域是怎样划分的。

Spark 内存区域划分

对于任意一个 Executor 来说，Spark 会把内存分为 4 个区域，分别是 Reserved Memory、User Memory、Execution Memory 和 Storage Memory。



极客时间

Spark内存区域划分

其中，Reserved Memory 固定为 300MB，不受开发者控制，它是 Spark 预留的、用来存储各种 Spark 内部对象的内存区域；User Memory 用于存储开发者自定义的数据结构，例如 RDD 算子中引用的数组、列表、映射等等。

Execution Memory 用来执行分布式任务。分布式任务的计算，主要包括数据的转换、过滤、映射、排序、聚合、归并等环节，而这些计算环节的内存消耗，统统来自于 Execution Memory。

Storage Memory 用于缓存分布式数据集，比如 RDD Cache、广播变量等等。关于广播变量的细节，我们留到第 10 讲再去展开。RDD Cache 指的是 RDD 物化到内存中的副本。在一个较长的 DAG 中，如果同一个 RDD 被引用多次，那么把这个 RDD 缓存到内存中，往往会大幅提升作业的执行性能。我们在这节课的最后会介绍 RDD Cache 的具体用法。

不难发现，Execution Memory 和 Storage Memory 这两块内存区域，对于 Spark 作业的执行性能起着举足轻重的作用。因此，**在所有的内存区域中，Execution Memory 和 Storage Memory 是最重要的，也是开发者最需要关注的。**

在 Spark 1.6 版本之前，Execution Memory 和 Storage Memory 的空间划分是静态的，一旦空间划分完毕，不同内存区域的用途与尺寸就固定了。也就是说，即便你没有缓

存任何 RDD 或是广播变量，Storage Memory 区域的空闲内存也不能用来执行映射、排序或聚合等计算任务，宝贵的内存资源就这么白白地浪费掉了。

考虑到静态内存划分的弊端，在 1.6 版本之后，Spark 推出了**统一内存管理模式**，在这种模式下，Execution Memory 和 Storage Memory 之间可以相互转化。这是什么意思呢？接下来，我们一起走进斯巴克集团分公司，看看不同内存区域相互转化的逻辑。

不同内存区域的相互转化

刚一走进分公司的大门，我们就能看到工人们在工地上如火如荼的忙碌景象。走近一问，才知道他们承接了一个“集装箱改装活动房”的建筑项目。顾名思义，这个项目的具体任务，就是把集装箱改装成活动房。

活动房的制作过程并不复杂，只需一系列简单的步骤，就能把集装箱改装为小巧而又别致的活动房，这些步骤包括清洗、切割开窗、切割开门、刷漆、打隔断、布置家居、装饰点缀。活动房的制作在工地上完成，成功改装的活动房会被立即拉走，由货运卡车运往集团公司的物流集散地。

好了，介绍完集装箱改装活动房的项目，我们必须要交代一下这个项目与 Spark 之间的关联关系。毕竟，再有趣的故事，也是用来辅助咱们更好地学习 Spark 嘛。

项目中涉及的原材料、施工步骤与 Spark 之间的类比关系，我把它整理到了下面的这张表格中：

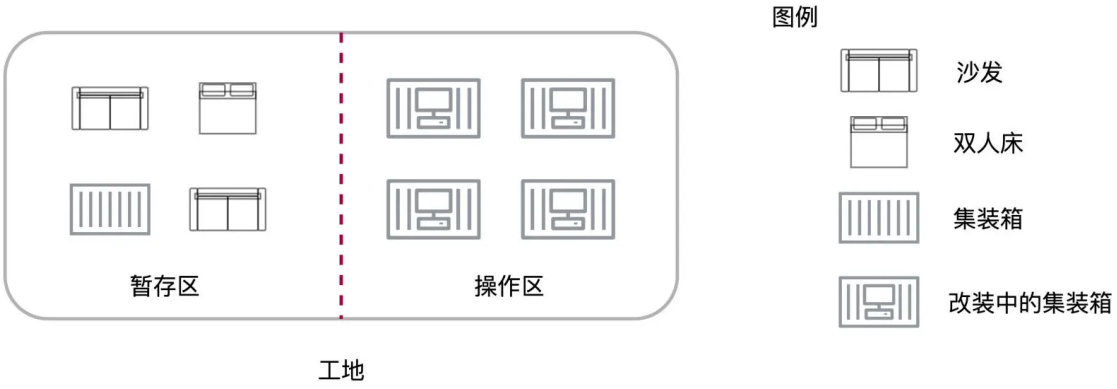
建筑项目	Spark
集装箱改装活动房的项目	Task所在Stage
集中箱、沙发、床	RDD数据源
施工步骤（如切割门窗等）	RDD算子
工地	节点内存

极客时间

建筑项目与Spark的类比

从表中可以看到，集装箱相当于是 RDD 数据源，而切割门窗等施工步骤，对应的正是各式各样的 RDD 算子。而工地用于提供施工场所，这与计算节点内存提供数据处理场所的作用如出一辙。这么看下来，集装箱改装活动房的项目，就可以看作是 Spark 作业，或者说是 Spark 应用。

接下来，我们来考察一下这个项目的施工过程。走近工地，我们发现工地上赫然划着一条红色的虚线，把工地一分为二。虚线的左侧，堆放着若干沾满泥土的集装箱，而工地的右侧，则是工人们在集装箱上叮叮当当地做着改装，有的集装箱已经开始布置家居，有的还在切割门窗。



极客时间

施工工地示意图

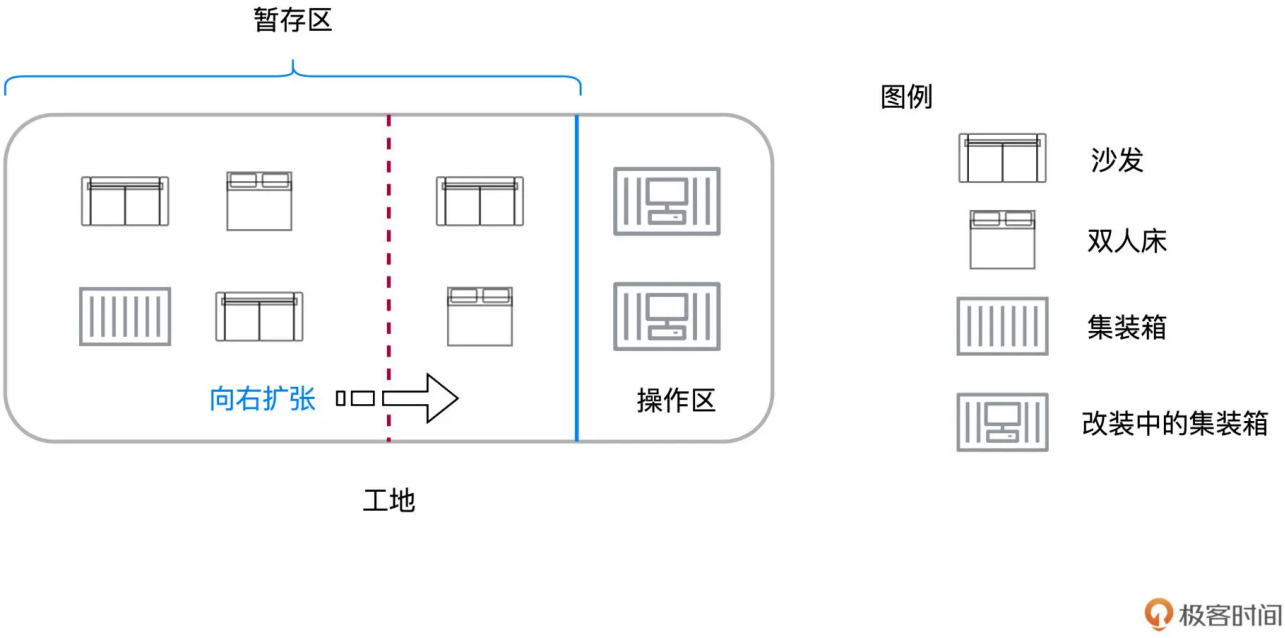
看到地上的红线，我们不免好奇，走近前去问，工头为我们道清了原委。

按理说，像集装箱、家具这些生产资料都应该放在临时仓库（节点硬盘）的，工地（节点内存）原则上只用来进行改装操作。不过，工地离临时仓库还有一段距离，来回运输不太方便。

为了提升工作效率，工地被划分成两个区域。在上图中，红线左边的那块地叫作暂存区（Storage Memory），专门用来暂存建筑材料；而右边的那部分叫作操作区（Execution Memory），用来给工人改装集装箱、制作活动房。

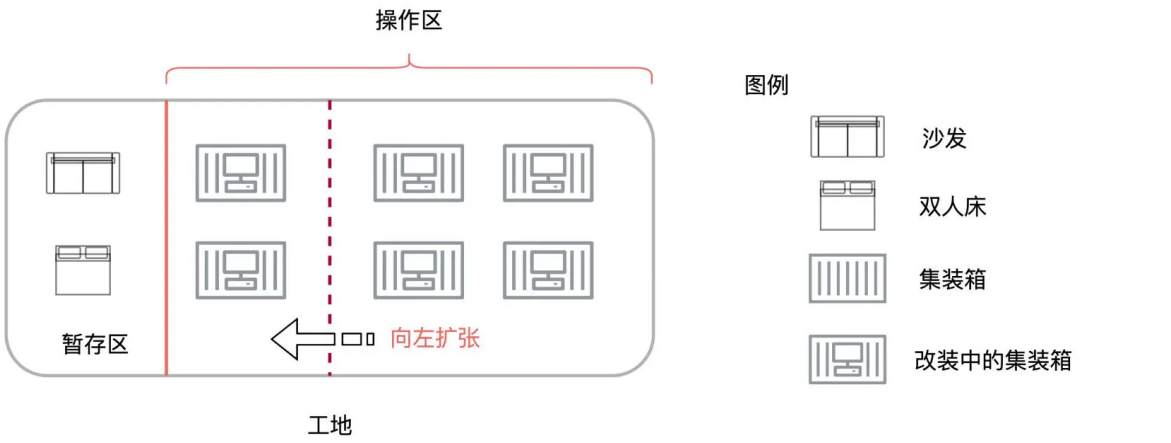
之所以使用虚线标记，原因就在于，两块区域的尺寸大小并不是一成不变的，当一方区域有空地时，另一方可以进行抢占。

举例来说，假设操作区只有两个工人（CPU 线程）分别在改装集装箱，此时操作区空出来可以容纳两个物件的空地，那么这片空地就可以暂时用来堆放建筑材料，暂存区也因此得到了实质性的扩张。



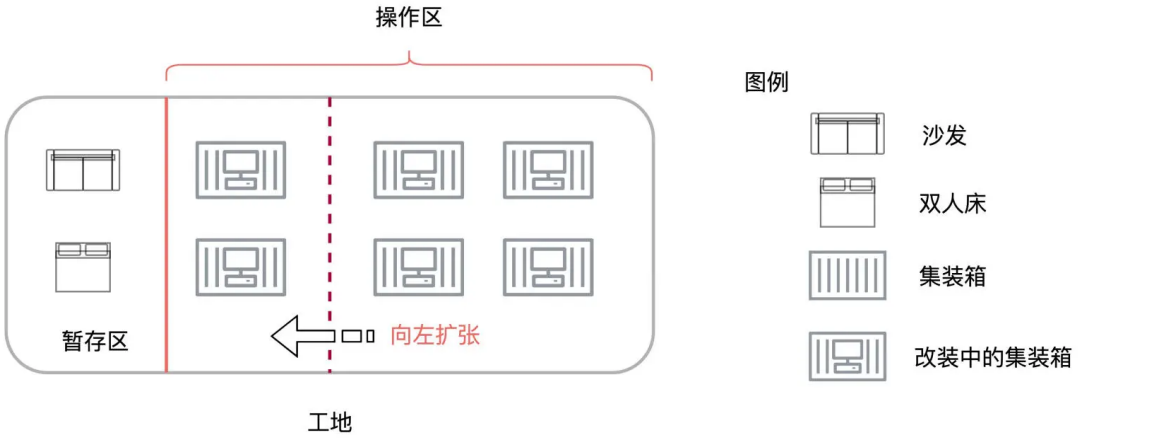
抢占模式1：暂存区抢占操作区

不过，当有足够的工人可以扩大生产的时候，比如在原有两个工人在作业的基础上，又来了两个工人，此时共有 4 个工人可以同时制作活动房，那么红色虚线到蓝色实线之间的任何物件（比如上图的沙发和双人床），都需要腾出到临时仓库，腾空的区域交给新来的两个工人改装集装箱。毕竟，改装集装箱、制作活动房，才是项目的核心任务。



抢占模式2：操作区抢占暂存区

相反，如果暂存区堆放的物件比较少、留有空地，而工人又比较充裕，比如有 6 个工人可以同时进行改装，那么此时暂存区的空地就会被操作区临时征用，给工人用来制作活动房。这个时候，操作区实际上也扩大了。



抢占模式2：操作区抢占暂存区

当有更多的物件需要堆放到暂存区的时候，扩张的操作区相应地也需要收缩到红色虚线的位置。不过，对于红色实线与红色虚线之间的区域，我们必须要等到工人们把正在改装的活动房制作完毕（Task Complete），才能把这片区域归还给暂存区。

好啦，活动房的项目到这里就介绍完了。不难发现，操作区类比的是 Execution Memory，而暂存区其实就是 Storage Memory。Execution Memory 和 Storage Memory 之间的抢占规则，一共可以总结为 3 条：

如果对方的内存空间有空闲，双方可以互相抢占；

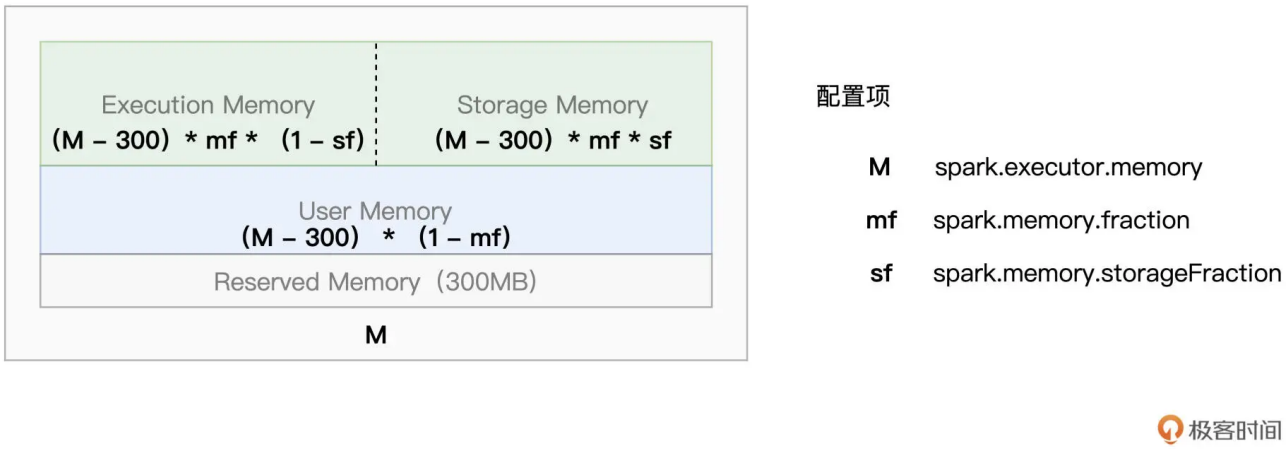
对于 Storage Memory 抢占的 Execution Memory 部分，当分布式任务有计算需要时，Storage Memory 必须立即归还抢占的内存，涉及的缓存数据要么落盘、要么清除；

对于 Execution Memory 抢占的 Storage Memory 部分，即便 Storage Memory 有收回内存的需要，也必须要等到分布式任务执行完毕才能释放。

介绍完 Execution Memory 与 Storage Memory 之间的抢占规则之后，接下来，我们来看看不同内存区域的初始大小是如何设置的。

内存配置项

总体来说，Executor JVM Heap 的划分，由图中的 3 个配置项来决定：



内存区域相关配置项

其中 spark.executor.memory 是**绝对值**，它指定了 Executor 进程的 **JVM Heap 总大小**。另外两个配置项，spark.memory.fraction 和 spark.memory.storageFraction 都是**比例值**，它们指定了划定不同区域的**空间占比**。

spark.memory.fraction 用于标记 Spark 处理分布式数据集的内存总大小，这部分内存包括 Execution Memory 和 Storage Memory 两部分，也就是图中绿色的矩形区域。（M – 300）*（1 – mf）刚好就是 User Memory 的区域大小，也就是图中蓝色区域的部分。

spark.memory.storageFraction 则用来进一步区分 Execution Memory 和 Storage Memory 的初始大小。我们之前说过，Reserved Memory 固定为 300MB。（M – 300）* mf * sf 是 Storage Memory 的初始大小，相应地，（M – 300）* mf *（1 – sf）就是 Execution Memory 的初始大小。

熟悉了以上 3 个配置项，作为开发者，我们就能有的放矢地去调整不同的内存区域，从而提升内存的使用效率。我们在前面提到，合理地使用 RDD Cache 往往能大幅提升作业的执行性能，因此在这一讲的最后，我们一起来学习一下 RDD Cache 的具体用法。

RDD Cache

在一个 Spark 作业中，计算图 DAG 中往往包含多个 RDD，我们首先需要弄清楚什么时候对哪个 RDD 进行 Cache，盲目地滥用 Cache 可不是明智之举。我们先说结论，**当同一个**

RDD 被引用多次时，就可以考虑对其进行 Cache，从而提升作业的执行效率。

我们拿第 1 讲中的 Word Count 来举例，完整的代码如下所示：

复制代码

```
import org.apache.spark.rdd.RDD

val rootPath: String = _
val file: String = s"${rootPath}/wikiOfSpark.txt"

// 读取文件内容
val lineRDD: RDD[String] = spark.sparkContext.textFile(file)

// 以行为单位做分词
val wordRDD: RDD[String] = lineRDD.flatMap(line => line.split(" "))
val cleanWordRDD: RDD[String] = wordRDD.filter(word => !word.equals(""))

// 把RDD元素转换为（Key，Value）的形式
val kvRDD: RDD[(String, Int)] = cleanWordRDD.map(word => (word, 1))

// 按照单词做分组计数
val wordCounts: RDD[(String, Int)] = kvRDD.reduceByKey((x, y) => x + y)

// 打印词频最高的5个词汇
wordCounts.map{case (k, v) => (v, k)}.sortByKey(false).take(5)

// 将分组计数结果落盘到文件
val targetPath: String = _
wordCounts.saveAsTextFile(targetPath)
```

细心的你可能发现了，我们今天的代码，与第 1 讲中的代码实现不同。我们在最后追加了 saveAsTextFile 落盘操作，这样一来，wordCounts 这个 RDD 在程序中被引用了两次。

如果你把这份代码丢进 spark-shell 去执行，会发现 take 和 saveAsTextFile 这两个操作执行得都很慢。这个时候，我们就可以考虑通过给 wordCounts 加 Cache 来提升效率。

那么问题来了，Cache 该怎么加呢？很简单，你只需要在 wordCounts 完成定义之后，在这个 RDD 之上依次调用 cache 和 count 即可，如下所示：

复制代码

```
// 按照单词做分组计数
val wordCounts: RDD[(String, Int)] = kvRDD.reduceByKey((x, y) => x + y)
```

```
3
4 wordCounts.cache// 使用cache算子告知Spark对wordCounts加缓存
5 wordCounts.count// 触发wordCounts的计算，并将wordCounts缓存到内存
6
7 // 打印词频最高的5个词汇
8 wordCounts.map{case (k, v) => (v, k)}.sortByKey(false).take(5)
9
10 // 将分组计数结果落盘到文件
11 val targetPath: String = _
12 wordCounts.saveAsTextFile(targetPath)
```

由于 cache 函数并不会立即触发 RDD 在内存中的物化，因此我们还需要**调用 count 算子来触发这一执行过程**。添加上面的两条语句之后，你会发现 take 和 saveAsTextFile 的运行速度明显变快了很多。强烈建议你在 spark-shell 中对比添加 Cache 前后的运行速度，从而直观地感受 RDD Cache 对于作业执行性能的提升。

在上面的例子中，我们通过在 RDD 之上调用 cache 来为其添加缓存，而在背后，**cache 函数实际上会进一步调用 persist（MEMORY_ONLY）来完成计算**。换句话说，下面的两条语句是完全等价的，二者的含义都是把 RDD 物化到内存。

复制代码

```
1 wordCounts.cache
2 wordCounts.persist(MEMORY_ONLY)
```

就添加 Cache 来说，相比 cache 算子，persist 算子更具备普适性，结合多样的存储级别（如这里的 MEMORY_ONLY），persist 算子允许开发者灵活地选择 Cache 的存储介质、存储形式以及副本数量。

Spark 支持丰富的存储级别，每一种存储级别都包含 3 个最基本的要素。

存储介质：数据缓存到内存还是磁盘，或是两者都有

存储形式：数据内容是对象值还是字节数组，带 SER 字样的表示以序列化方式存储，不带 SER 则表示采用对象值

副本数量：存储级别名字最后的数字代表拷贝数量，没有数字默认为 1 份副本。

我把 Spark 支持的存储级别总结到了下表，其中打钩的地方，表示某种存储级别支持的存储介质与存储形式，你不妨看一看，做到心中有数。

存储级别	存储介质		存储形式		副本数量	备注
	内存	磁盘	对象值	序列化		
MEMORY_ONLY	✅		✅		1	
MEMORY_ONLY_2	✅		✅		2	
MEMORY_ONLY_SER	✅			✅	1	
MEMORY_ONLY_SER_2	✅			✅	2	
DISK_ONLY		✅		✅	1	
DISK_ONLY_2		✅		✅	2	
DISK_ONLY_3		✅		✅	3	
MEMORY_AND_DISK	✅	✅	✅	✅	1	内存的部分以对象值存储，磁盘部分序列化
MEMORY_AND_DISK_2	✅	✅	✅	✅	2	
MEMORY_AND_DISK_SER	✅	✅		✅	1	内存和磁盘都以序列化的字节数组形式进行存储
MEMORY_AND_DISK_SER2	✅	✅		✅	2	

极客时间

RDD Cache的存储级别

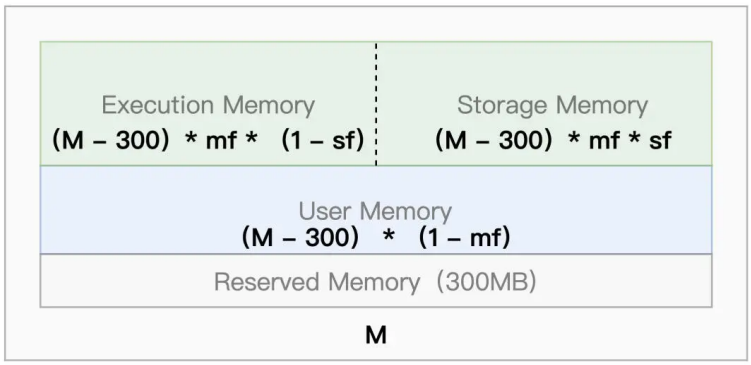
通过上表对琳琅满目的存储级别进行拆解之后，我们就会发现，它们不过是存储介质、存储形式和副本数量这 3 类基本要素的排列组合而已。上表列出了目前 Spark 支持的所有存储级别，通过它，你可以迅速对比查找不同的存储级别，从而满足不同的业务需求。

重点回顾

今天这一讲，你需要掌握 Executor JVM Heap 的划分原理，并学会通过配置项来划分不同的内存区域。

具体来说，Spark 把 Executor 内存划分为 4 个区域，分别是 Reserved Memory、User Memory、Execution Memory 和 Storage Memory。

通过调整 spark.executor.memory、spark.memory.fraction 和 spark.memory.storageFraction 这 3 个配置项，你可以灵活地调整不同内存区域的大小，从而去适配 Spark 作业对于内存的需求。



配置项

M	spark.executor.memory
mf	spark.memory.fraction
sf	spark.memory.storageFraction

极客时间

内存区域相关配置项

再者，在统一内存管理模式下，Execution Memory 与 Storage Memory 之间可以互相抢占，你需要弄清楚二者之间的抢占逻辑。总结下来，内存的抢占逻辑有如下 3 条：

如果对方的内存空间有空闲，双方可以互相抢占；

对于 Storage Memory 抢占的 Execution Memory 部分，当分布式任务有计算需要时，Storage Memory 必须立即归还抢占的内存，涉及的缓存数据要么落盘、要么清除；

对于 Execution Memory 抢占的 Storage Memory 部分，即便 Storage Memory 有收回内存的需要，也必须要等到分布式任务执行完毕才能释放。

最后，我们介绍了 RDD Cache 的基本用法，当一个 RDD 在代码中的引用次数大于 1 时，你可以考虑通过给 RDD 加 Cache 来提升作业性能。具体做法是在 RDD 之上调用 cache 或是 persist 函数。

其中 persist 更具备普适性，你可以通过指定存储级别来灵活地选择 Cache 的存储介质、存储形式以及副本数量，从而满足不同的业务需要。

每课一练

好啦，这节课就到这里了，我们今天的练习题是这样的：

给定如下配置项设置，请你计算不同内存区域（Reserved、User、Execution、Storage）的空间大小。

配置项	参数设置
spark.executor.memory	10GB
spark.memory.fraction	0.8
spark.memory.storageFraction	0.4

极客时间

欢迎你在评论区分享你的答案。如果这一讲对你有帮助，也欢迎你把这一讲分享给自己的朋友，我们下一讲再见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 07 | RDD常用算子（二）：Spark如何实现数据聚合？

下一篇 09 | RDD常用算子（三）：数据的准备、重分布与持久化

更多学习推荐

精选留言 (5)

写留言

AIK
2021-09-27

R:300M
U:(10240 - 300) * (1 - 0.8) =1988
E:(10240 - 300) * 0.8 * (1 - 0.4) = 4771.2
S:(10240 -300) * 0.8 * 0.4 = 3180.8
老师实际生产环境，把spark. executor. memory设置为物理内存的80%合理吗？

作者回复: 满分💯

可以的，这里的0.8，其实不是物理内存的80%，而是分配给Executor JVM heap的80%，没什么风险的

4

Geek_2dfa9a
2021-09-30

这里粗略估计1GB为1000MB
Reserved Memory = 300MB
Execution Memory + Storage Memory = (10GB - 0.3GB) * 0.8 = 9.7GB * 0.8 = 7.76GB
User Memory = 9.7GB - 7.76GB = 1.94GB

Execution Memory = 7.76GB * (1 - 0.4) = 7.76GB * 0.6 = 4.66GB…

展开

作者回复: 正解，满分💯

井先生
2021-09-27

既然user memory存储用户自定义的数据结构，我在纯sql的spark job中是不是把spark.memory.fraction设置很大的值，只给user memory留比较小的空间以提高execution和storage memory的大小，进而提高内存的使用率呢

展开

作者回复: 没错，正解～ 老弟思考的相当到位～

大志
2021-09-27

error: not found: value MEMORY_ONLY

import org.apache.spark.storage.StorageLevel._
wordCounts.persist(MEMORY_ONLY) // 就可以了

展开

作者回复: 正解～

1

Geek_796a41
2021-09-27

老师，我想问下spark.memory.fraction 和 spark.memory.storageFraction这两个值在设置的时候都有什么参考？

作者回复: 可以参考后面的配置项那一讲（12讲）哈～ 应该很快就更新啦！敬请关注～